Performance Improvement Technique in Column-Store

Tejaswini Apte[1], Dr. Maya Ingale[2], Dr. A.K. Goyal[2]
Sinhgad Institute of Business Administration & Research, Kondhwa(Bk.),Pune- 411048,India[1]
apte.tejaswini@gmail.com [1]
Devi Ahilya Vishwa Vidyalaya,Indore, India[2]
{maya_ingle@rediffmail.com , goyalkcg@yahoo.com}[2]

***Abstract*: Column-oriented database has gained popularity as "Data Warehousing" data and performance issues for "Analytical Queries" have increased. Each attribute of a relation is physically stored as a separate column, which will help analytical queries to work fast. The overhead is incurred in tuple reconstruction for multi attribute queries. Each tuple reconstruction is joining of two columns based on tuple IDs, making it significant cost component. For reducing cost, physical design have multiple presorted copies of each base table, such that tuples are already appropriately organized in different orders across the various columns.**

This paper proposes a novel design, called partitioning, that minimizes the tuple reconstruction cost. It achieves performance similar to using presorted data, but without requiring the heavy initial presorting step. In addition, it handles dynamic, unpredictable workloads with no idle time and frequent updates. Partitioning provides the direct loading of the data in respective partitions. Partitions are created on the fly and depend on distribution of data, which will work nicely in limited storage space environments.

***General Terms:* Algorithms, Performance, Design**

***Keywords:* Database Cracking, Self-organization**

I. Introduction

The main feature of column-stores is to provide improved performance over row store for analytical queries, which contains few attribute of relation R. Position based joining, has done for queries for tuple reconstruction, which required multiple attributes [2, 6, 10]. For each relation Rj in query q, a column store has to perform Nj-1 tuple reconstruction operations for Rj within q. Recent study has shown tuple reconstruction in two ways [2]. In early tuple reconstruction at each point at which a column is accessed, add the column to an intermediate tuple representation, if that column is needed by some later operator or is included in the set of output columns. At the top of the query plan, these intermediate tuples can be directly output to the user. Early materialization is not always the best strategy to employ in a column store. As early materialization reads all the blocks from disk, stitch them together and then apply the selection operator.

In later strategy i.e. late tuple construction has no tuple construction until some part of the plan has been processed. It scans the relation recursively by applying the predicate on it It has the property to reaccess all the relation and extract the satisfying records by position wise join. This approach can potentially be more CPU efficient because it requires fewer intermediate tuples to be stitched together (which is a relatively expensive operation as it can be thought of as a join on position), and position lists are small, highly-compressible data structures that can be operated on directly with very little overhead. For example, 32 (or 64 depending on processor word size) positions can be intersected at once when ANDing together two position lists represented as bit-strings.

Prior research has suggested that important optimizations specific to column-oriented DBMSs include:

Late materialization: (when combined with the block iteration optimization below, this technique is also known as vectorized query processing [9, 25]), where columns read off disk are joined together into rows as late as possible in a query plan [5].

Column-specific compression: techniques, run-length encoding, with direct operation on compressed data when using late-materialization plans [4].

Invisible joins: which substantially improves join performance in late-materialization column stores, especially on the types of schemas found in data warehouses [28].

Partial sideways cracking [29]: that minimizes the tuple reconstruction cost in a self-organizing way. It achieves performance similar to using presorted data, but without requiring the heavy initial presorting step it.

The Column-store database has dynamic tuple reconstruction for multi attribute queries, which has multi block reads. Partitioning the data and maintaining indexes for block address, record keys, column value and partition would enable faster access to multiple blocks. The specific structure of index files would depend on read/write requirements of a specific application. Optimizer can determine such index and partition requirements dynamically. Intelligent sequencing of disk seek operations will enable reading data from one block in one sweep. We define the partitioning over the relation R for the column, which has used highly in aggregate queries. Partitioning has to done dynamically, as soon as new ranges for the partition column appear. Column store has to maintain the Index on partition for searching partition

II. BACKGROUND AND PRIOR WORK

This section briefly presents related efforts to characterize column-store performance relative to traditional row-stores. Although the idea of vertically partitioning database tables to improve performance has been around a long time [1, 7, 16], the MonetDB [10] and the MonetDB/X100 [9] systems pioneered the design of modern column-oriented database systems and vectorized query execution. They show that column-oriented designs –due to superior CPU and cache performance (in addition to reduced I/O) – can dramatically outperform commercial and open source databases on benchmarks like TPC-H. The MonetDB work does not, however, attempt to evaluate what kind of performance is possible from row-stores using column-oriented techniques, and to the best of our knowledge, their optimizations have never been evaluated in the same context as the C-Store optimization of direct operation on compressed data.

The fractured mirrors approach [21] is another recent column store system, in which a hybrid row/column approach is proposed. Here, the row-store primarily processes updates and the column store primarily processes reads, with a background process migrating data from the row-store to the column-store. This work also explores several different representations for a fully vertically partitioned strategy in a row-store, concluding that tuple overheads in a naive scheme are a significant problem, and that prefetching of large blocks of tuples from disk is essential to improve tuple reconstruction times.

C-Store [22] is a more recent column-oriented DBMS. It includes many of the same features as MonetDB/X100, as well as optimizations for direct operation on compressed data [4]. Like the other two systems, it shows that a column-store can dramatically outperform a row-store on warehouse workloads, but doesn't carefully explore the design space of feasible row-store physical designs. In this paper, we dissect the performance of C-Store, noting how the various optimizations proposed in the literature (e.g., [4, 5]) contribute to its overall performance relative to a row-store on a complete data warehousing benchmark, something that prior work from the C-Store group has not done.

Harizopoulos et al. [14] compare the performance of a row and column store built from scratch; studying simple plans that scan data from disk only and immediately construct tuples ("early materialization"). This work demonstrates that in a carefully controlled environment with simple plans, column stores outperform row stores in proportion to the fraction of columns they read from disk, but doesn't look specifically at optimizations for improving row-store performance, or at some of the advanced techniques for improving column-store performance.

Halverson et al. [13] built a column-store implementation in Shore and compared an unmodified (row-based) version of Shore to a vertically partitioned variant of Shore. Their work proposes an optimization; called "super tuples" that avoids duplicating header information and batches many tuples together in a block, which can reduce the overheads of the fully vertically partitioned scheme and which, for the benchmarks included in the paper, make a vertically partitioned database competitive with a column-store. The paper does not, however, explore the performance benefits of many recent column-oriented optimizations, including a variety of different compression methods or late-materialization. Nonetheless, the "super tuple" is the type of higher-level optimization that this paper concludes will be needed to be added to row-stores in order to simulate column-store performance.

D. B. Abadi [27] built a invisible join over column oriented database which substantially improves join performance in late-materialization column stores, especially on the types of schemas found in data warehouses.

Stratos Idreos, Martin L. Kersten, Stefan Manegold [29] partial sideways cracking, that minimizes the tuple reconstruction cost in a self-organizing way. It achieves performance similar to using presorted data, but without requiring the heavy initial presorting step it.

III. PARTITION TRANSACTION

This section demonstrates the partitioning algorithm, which has to execute for moving data into specific partition.

Partition Relation R :

```
R=Pick (Relation)
Indexmap valuemap
If R=emptyset
R=new File [page size]
Load data
Go to Label 1
End If

Label 1:
Count=0
ValuesCount = number of values in the partition column
of Relation
P=pick[R & ValuesCount]
Sort the Relation
Lval=Read the last value of range partition

While (I < P)
If new value >Lval then
P[I] = new File [Page Size]
End if
Write(P[I],value]
Valuemap.put(value,count)
Write (Idx[P[I]],valuemap)
I++
Count++
```

IV. PARTITION MAP

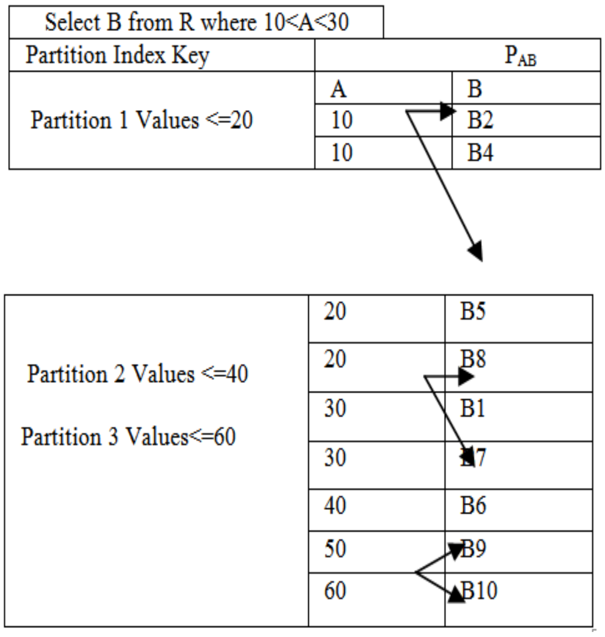
This section demonstrates how partition enables a column-store to efficiently handle multi-attribute queries. It achieves similar performance to presorted data but without the heavy initial cost and the restrictions on updates. We define a partition key index map as a two column table, with first column contains the index key (same as partition key) and second column contains address of the partition. Database Optimizer will work through following steps:

```
1. Read (Idx[P[I] ],valuemap)
2. Append Data:
        Buffer *pBuffer,
        Byte *pByte,
        Int datalength

        {
          while (more input partition data) {
            fill the current memory block;
            if (more input data) {
              allocate a new memory block and add it into
        the linked list;
            }
          }
        }

3. Read partition values from memory:
    Cursor.values= P[I]
        If Cursor.currentval=0 then 'No more values in
        stack'
        Else
         Pop[Cursor.currentval]
         Nextval=Currentval[I]-1
        Currentval=Nextval
        End IF
```

Example. Assume a relation R (A; B) shown in Figure 1. The first query requests values of B where a restriction on A holds. The system searches the partition according to range specified in a query and picks the partition based on selection predicate.



Select B from R where 10<A<30

Partition Index Key	P_{AB}	
Partition 1 Values <=20	A	B
	10	B2
	10	B4

Partition 2 Values <=40	20	B5
	20	B8
	30	B1
Partition 3 Values<=60	30	B7
	40	B6
	50	B9
	60	B10

Figure 1: A simple example

Via Partitioning the qualifying B values are already clustered together aligned with the qualifying A values. The required partition loads into memory by reducing the disk seeks and memory requirement.

V. CONCLUSION

In this paper, we introduce partitioning, a key component in column-store based on physical organization of data for performance optimization for tuple reconstruction. It enables efficient processing of complex multi-attribute queries by minimizing the costs of late tuple reconstruction, achieving performance competitive with using presorted data. Database partitioning has only scratched the surface of this promising direction for column store DBMSs. The research agenda includes calls for innovations on compression as well as optimization strategies, e.g., cache-conscious partition size enforcement.

VI. REFERENCES

[1]http://www.sybase.com/products/informationmanagement/sybaseiq.

[2]TPC-H Result Highlights Scale 1000GB.http://www.tpc.org/tpch/results/tpch result detail.asp?id=107102903.

[3] D. J. Abadi. Query execution in column-oriented database systems. MIT PhD Dissertation, 2008. PhD Thesis.

[4] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In SIGMOD, pages 671–682, 2006.

[5] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In ICDE, pages 466–475, 2007.

[6] Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In VLDB, pages 169–180, 2001.

[7] D. S. Batory. On searching transposed files. ACM Trans. Database Syst., 4(4):531–544, 1979.

[8] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. J. ACM, 28(1):25–40, 1981.

[9] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In CIDR, 2005.

[10]P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. VLDB Journal, 8(2):101–119, 1999.

[11]G. Graefe. Volcano - an extensible and parallel query evaluation system. 6:120–135, 1994.

[12]G. Graefe. Efficient columnar storage in b-trees. SIGMOD Rec., 36(1):3–6, 2007.

[13]Halverson, J. L. Beckmann, J. F. Naughton, and D. J. Dewitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University of Wisconsin-Madison, 2006.

[14]S. Harizopoulos, V. Liang, D. J. Abadi, and S. R. Madden. Performance tradeoffs in read-optimized databases. In VLDB, pages 487–498, 2006.

[15]S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: a simultaneously pipelined relational query engine. In SIGMOD, pages 383–394, 2005.

[16]S. Khoshafian, G. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In ICDE, pages 636–643, 1987.

[17]P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. SIGMOD Rec., 24(3):8–11, 1995.

[18]P. E. O'Neil, X. Chen, and E. J. O'Neil. Adjoined Dimension Column Index (ADC Index) to Improve Star Schema Query Performance. In ICDE, 2008.

[19]P. E. O'Neil, E. J. O'Neil, and X. Chen. The Star Schema Benchmark (SSB). http://www.cs.umb.edu/_poneil/StarSchemaB.PDF.

[20]S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In ICDE, 2001.

[21]R. Ramamurthy, D. Dewitt, and Q. Su. A case for fractured mirrors. In VLDB, pages 89 – 101, 2002.

[22]M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E.O'Neil, A. Rasin, N. Tran, and S. B.Zdonik. C-Store: A Column-Oriented DBMS. In VLDB, pages 553–564, 2005.

[23]Weininger. Efficient execution of joins in a star schema. In SIGMOD, pages 542–545, 2002.

[24]J. Zhou and K. A. Ross. Buffering databse operations for enhanced instruction cache performance. In SIGMOD, pages 191–202, 2004.

[25]M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - A DBMS In The CPU Cache. IEEE Data Engineering Bulletin, 28(2):17–22, June 2005.

[26]M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In ICDE, 2006.

[27]D. J. Abadi, Samuel R. Madden, Nabil Hachem: ColumnStores vs. RowStores: How Different Are They Really?

[28]Hasso Plattner: A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database

[29]Stratos Idreos, Martin L. Kersten, Stefan Manegold: Selforganizing Tuple Reconstruction in Columnstores